Module Interconnection Frameworks for a Real-Time Spreadsheet

Progress Report
July 29, 1993

Program: SBIR N92-112

Scientific Officer: Ralph Wachter, Office of Naval Research

Principal Investigator: Richard Clarke, RTware, Inc.

1. Overview

RTware is working on a proposal for an SBIR Phase II software research and development project for distributed control systems (DCS) using Module Interconnect Frameworks (MIF). SBIR Phase I work includes research into the feasibility, design and benefits of such a system, resulting in a detailed Phase II proposal. The proposed software package will be based on RTware's ControlCalc real-time spreadsheet control system and the Polylith MIF system from the University of Maryland. Work in the initial stage of Phase I has identified the main outlines of the package, and described key benefits to the Department of Defense (see the May 30th Progress Report). Work in the latest time period involved developing a detailed description of the proposed system and trying test implementations with the Polylith package. That description breaks the system down into two major components: The first component (DCS-I) is a distributed services capability for accessing remote procedures and data both into and out of the ControlCalc spreadsheet. The second component (DCS-II) will allow the largely transparent distribution of compiled ControlCalc code segement across remote, and heterogeneous, platforms. The specification of DCS-1 is essentially complete and is presented in this report. The specification of DCS-II is still in the conceptual stage. The concepts and issues under consideration are presented in this report. A detailed specification of DCS-II is scheduled for inclusion in the next progress report.

2. Work in Progress Description

The latest phase of work has focused on developing a detailed functional specification for a Polylith-based MIF distributed control system, and validating that specification with Polylith test programs. A major part of the specification describes a distributed services registration facility. That facility will provide both an API (application program interface) for interactive programs and a set of utilities for manual (text edit) interface. Another major part of the specification describes a spreadsheet interface for both providing and accessing distributed services. The use of Polylith to implement these concepts allows mixed language distributed services, since the service registration facility will generate a Polylith Module Interconnect Language (MIL) file. Polylith already includes a 'C' language API. The spreadsheet interface can be considered an API to the Polylith bus for spreadsheet programmers.

RTware has also been working with the Naval Research Laboratories on a project which will be able to serve as an excellent demonstration site for the resulting distributed control system. This project was recently awarded to RTware, with the selection of ControlCalc as the control language and GUI system. The project is the design of the next generation of radar control and operator interface for the Orion "sub-hunter" aircraft. RTware has discussed the possibility of using the project's laboratory as a test site for the proposed ONR MIF control system with the responsible engineer (Sharon Hrin) on the radar project. Her response has been very positive. Within her lab, there are a large number of heterogenous systems which often must work together in distributed applications. With the radar project, for example, up to eight pods will be controlled simultaneously. Furthermore, the radar system will also be running in a laboratory simulation environment in which computers doing simulation and data analysis are interacting with the control system. A section below describes the lab's requirements in more detail.

A preliminary functional specification is presented in this report describing a distributed service and remote access interface for the spreadsheet (referred to as DCS-I). The phase II proposal will also include the ability to distribute compiled spreadsheet code segments among different processors (DCS-II). This report describes the major technical issues involved in this concept, but does not present a functional specification. The NRL group is particularly interested in distributed code segments running on attached DSP boards. RTware is therefore investigating the SPOX DSP operating system, which is ported to most commercially available DSP boards.

Finally, in the last month RTware has released a ControlCalc spreadsheet interface to the Dataviews graphical environment, a well-established graphics package for the major UNIX and minicomputer platforms. This work has demonstrated the effectiveness of ControlCalc's spreadsheet paradigm, and provides a distributed graphical front-end running on X-windows systems. With ControlCalc's multi-tasking spreadsheet structure controlling visual input and output objects, the need for procedural programming with the complex Dataviews API has been eliminated, without a significant loss of functionality.

3. Discussion in Detail

3.1. Distributed Control System I: Remote Access - Functional Specification

3.1.1. Overview

The DCS-I functional specification describes two fundamental capabilities:

1) Presenting spreadsheet services and data to external processes.
2) Calling remote modules from within the spreadsheet.

Both of these capabilities can be implemented using the Polylith, and in fact make up the core of what Polylith does. There are two problems that have to be addressed, however. First, there must be a way of defining services, or interfaces, from within a ControlCalc spreadsheet. Second, there must be functions that let the spreadsheet programmers create calls to such services in their spreadsheet templates. Note that in ControlCalc a running spreadsheet template is a set of tasks processing the shared memory data space of a three-dimensional spreadsheet. Each task is a separate executing process, instantiating a set of spreadsheet functions. The spreadsheet function set is therefore considered a simple functional language.

To understand the requirements of the DCS system, it is useful to understand the general sequence of operations in the implemention of a distributed application with Polylith. Distributed Polylith applications are constructed in a modular fashion. Each module consists of an executable file and a MIL "module_description" section file. The module_description section describes a "service" as an executable program (source or binary) and the host it runs on, along with its object attributes and the

Polylith interfaces the module will use (both input and output). Note that a node can declare multiple interfaces, and interfaces are declared as either input (sink or function) or output (source or client). An application consists of a set of module services instantiated by a master "bus" program that runs on each host involved. The bus programs requires that all the module_description MIL sections be linked together along with an application_description MIL section. The application_description section defines a node in the application, naming the node and specifying which module instantiates the node. Note that one module can be instantiated as multiple nodes, hence the declaration of node names mapped to module names. The application_description section also declares interface bindings by associating source/sink or client/function pairs. Each binding therefore maps two specific interfaces on two specific nodes. Once this is done, the bus program has all the routing information necessary for its operation. An application is actually executed by executing the bus program on each host, passing a parameter which is the linked MIL file. The bus program(s) then execute the module binaries, which can then use the interfaces.

Given this Polylith struture, any particular module can be created independently and used in multiple applications by simply rebinding its interfaces for each instantiation. This is the essence of a module interconnection framework.

3.1.2. Modular Interface to ControlCalc

Within the ControlCalc system, a module will be considered a particular spreadsheet template. There will be two form of interfaces: standard and user-defined. Standard interfaces will allow remote processes to perform such generic operations such as stopping and starting the control logic. User-defined interfaces will allow remote processes to read and write information from the control system spreadsheet, and optionally to trigger, pend and wait using the spreadsheets counting semaphore primitives. These modular interfaces will be either function or sink interfaces in Polylith, since they exist solely to respond to external requests. However, since control scans are the user-programmed executable functions in the spreadsheet, the services invoked by triggering a scan can be programmed to perform any function the spreadsheet is capable of, including accessing subsequent remote services, as described below.

3.1.2.1. Module implementation

The actual executable program implementing the template's interfaces will be named using the template file name with the extension .mod. This will be a 'C' function that will initialize by obtaining the interfaces it must support, and then loop continuously doing the Polylith bus function: mh_readselect. That function returns both a message buffer and a pointer to the name of the interface that the message came in on. The module program will then look up the corresponding definition for the message's interface. Each interface will have a list of spreadsheet data points which will be read or written. Interfaces will have a configuration option to synchronize with the real-time spreadsheet executing logic using the ControlCalc's existing counting semaphores. There will be a number modes of synchronization, so that explicit coding of the sequences is not required:

1) Monitor r/w control:	Wait for signal A, read and write data, send signal A
2) Monitor reads:	Wait for signal A, read data, send signal A
3) Monitor writes:	Wait for signal B, write data, send signal B
4) Monitor service:	Wait for signal C, do service in list below, send signal C
4) Signal Service:	Read/Write data, send signal A
5) Wait Service:	Wait for signal A, read/write data
6) Request service:	Write data, send signal A, wait for signal B, read data.
7) Monitored Request:	Wait for signal A, write data, send signal B, wait for signal C, read data, send signal A

These modes (Signal service and Wait service in particular) can be also be used to synchronize external processes, treating the spreadsheet logic as a rendevous mechanism. The Polylith interface will have no knowledge of this signalling involved, but reasonable naming of this type of interface can serve to document the purpose of services using these features. It is important in selecting these functions to be aware that some applications may use multiple instantiations of a module interfacing to one spreadsheet template. If this is anticipated, then it would be normal to use one of the monitor synchronization modes if interrupting an operation is not to be allowed. Note that the signals described abstractly as A,. B, and C will actually be a configurable ControlCalc signal number. There are 500 signal numbers available for synchronizing tasks that perform operations any one ControlCalc shared memory region. Since a template, and therefore the instantiations of the module interface, are associated with a shared memory region, these signals are common to all instantiations. This means that an interface with a monitor will allow only one instantiation to execute the monitored operation, but will not interfere with interfaces to another spreadsheet template, even though the same signal numbers may be used. On a UNIX system, this facility is implemented with SYS V IPC's, using a key number to allocate ranges of signal id numbers.

The ControlCalc spreadsheet is already a multi-tasking, shared memory system. Therefore the capabilities described above can be implemented quite easily. The spreadsheet data area already is kept in a shared memory segment as a sort of shared memory data base. A library of reentrant routines for reading and writing data, and sending and receiving signals, are already in place and used by programs such as I/O agents and GUI window agents that are a standard part of ControlCalc.

3.1.2.2. Data Types

Data types will be specified using the ControlCalc data types, which consist of either simple types (integers, booleans, floating point or strings) or arrays of simple types. An array type in the spreadsheet is a range, or rectangular section, of the spreadsheet. Structured types are not supported by the spreadsheet. There is therefore a corresponding data type in Polylith syntax for the data types in the spreadsheet. This will allow the generation of MIL language definitions directly from spreadsheet-style declarations. Other language interfaces can then access the MIL definitions without knowing about the spreadsheet data types (another design goal of MIF).

3.1.2.3. A Module Interface Definition Command in ControlCalc

In order to define these external interfaces, a module specification command will be added to ControlCalc's interactive command set. The module specification command will bring up a utility program that lets the user create, delete and modify interface definitions for the current template. These definitions will be saved as part of the template definition file. The user will also be able to cause a MIL file to be generated, and a new executable binary file for the module to be built. The MIL file will consist of a single "service" declaration, but without the host name field. The host name will be added by a later utility that builds the complete application. The command will be presented in an interactive window, presenting scrolling lists of input and output data points, radio button selectors for synchronization modes, text edit fields and push buttons for editing. The following commands/operations will be supplied:

Operations on a scrolling list of interfaces:

1) Add an interface.
2) Delete an interface.
3) Rename an interface.
4) Select an interface.

Operations on either of two scrolling list of data points:

1) Add
2) Delete
3) Edit
4) Verify data points

Mode Selections: Select one or none of the synchronization modes listed in section 3.1.2.1.
Signal number selection: Enter signal number to use for synchronization modes.

General Operations:

1) Accept new list
2) Discard changes to list
3) Create MIL file
4) Create module executable
5) View external interfaces (see section 3.1.4)

3.1.3. Defining an Application

Once a set of modules has been defined, an application is constructed by linking together the MIL files for each module interface, and adding the application definitions. The resulting file is a complete application MIL file, which is used by the bus program to start and run the application. The MIL files for each module are created by the templates (or programmers in other languages). These files are complete, except for the host names. Therefore linking these together means selecting and inserting the host names while concatenating the files together. The original files must be left untouched, so they can be regenerated when desired.

Declaring an application involves two types of declarations. The "tool" declaration instantiates a module as a node in the distributed application. The name of the node can be the name of the module which implements the node. It is also possible to instantiate multiple nodes using the same module. In that case, an explicit node name is placed in the tool declaration. Node names, of course must be unique. Once a tool is declared, then an instantiatation of an interface is defined by specifying node name-interface name pairs. Within a node, interface names also, of course, must be unique.

Finally, the interfaces are bound together by declaring mappings between input and output interfaces. At this point, the MIL file contains enough information to allow the bus program to start all the node programs and physically bind the interfaces together using whatever low-level transport structure the bus program implements. In current implementations that structure is always TCP/IP, and binding is done at the socket level. The actual process of passing input and output data back and forth is done by passing messages across the bindings, using the message formats that are defined in the interface declarations.

When the bus is run, two mode options are available, connection and keep alive. The default connect mode is to have all messages mediated by the bus program. Essentially all connections are made to the bus program, which acts as a router between the input and output half of every interface binding. Optionally, the direct connect mode allows connection directly between the node processes, which results in better performance. On the initial implementation, ControlCalc will use the standard mode. This is necessary in order to allow a single module to handle multiple interfaces to a spreadsheet template. Direct connect mode does not allow use of the mh_readselect option for monitoring multiple interfaces. In the general-purpose model we are implementing, a direct connect

option would require a single interface per node, or multiple nodes to implement a set of interfaces to one spreadsheet template. The speed benefit of direct connection may justify the overhead of multiple nodes, and is a question we will address when we benchmark the system.

The second option, keep alive, overrides the default I/O access mode, which is to open and close the I/O channel involved on every message transaction. In most real-time systems, keep-alive would be required, due to the very high overhead of opening and closing I/O channels. However, most UNIX implementations impose a relatively low limit (15 - 32) on the number of open I/O channels per process. Again, this limitation argues for a single node per interface, since the nodes' non-Polylithic interfaces to the ControlCalc spreadsheet is not limited in this manner, being based on shared memory and signals, not I/O channels. An argument can also be made in favor of letting the keep-alive option be applied per-binding, rather the per bus, as is the case in the current Polylith implemention. In that way, non-ControlCalc languages that are best thought of as nodes with many interfaces each could accept the performance loss, while performance critical bindings could keep their channels open.

3.1.3.1. Creating the application MIL file

Creating the application definition section of the MIL file is actually part programming of the application, since it contains statements which tie the module interfaces together, thereby instantiating modules However, this process is declarative programming, rather than sequential. It is therefore very amenable to a configuration utility front-end similar to the one described for the module interface definitions. Calling such a configuration utility within ControlCalc would also present a consistent interface style.

Programming and debugging issues also indicate the usefulness of a configuration utility. Using the construction technique for modules and their interfaces described in section 3.1.3., the result is a collection of MIL sections. In a complex system, this would mean a large number of small files, making it difficult to manage the application setup. An interactive configuration utility should therefore allow the user to display multiple sub-windows, each displaying a MIL module declaration section, to simplify browsing among the modules.

More importantly, both the application and the interface configuration utilities should provide interactive checking of syntax., preventing such obvious errors as:

1) Declaring duplicate node names.
2) Usng input interfaces in the output half of a bind declaration, and vice-versa.
3) Using unknown names.
4) Illegal syntax in general
5) Non-existence of hosts or executables

The current method of creating all the MIL file(s) is a text editor, followed by a compiler. Some typographical errors (such as naming mistakes) cannot be caught until run-time. An interactive utility would be able to provide immediate feedback, and browsing. Of course, errors involving unkown names should be posted as warnings, since it is not desirable to force, for example, definition of all interfaces before any bindings involving that interface. For a variety of reasons, we plan to have our application configuration utility maintain its own variation of a makefile which lists the modules involved without actually constructing a concatenated MIL file. However, all such files will be kept in text form, with a published syntax, so users who prefer a text editor, or have existing applications they want to add ControlCalc modules to, can do so without re-keying their application MIL section. Furthermore, the utility will be constructed as a stand-alone program, although a ControlCalc command option will be able to invoke it from within the spreadsheet.

The application configuration utility will provide the following operations:

1) Add/Delete modules (services)
2) Declare a host for a module
3) Add/Delete/Modify tools
4) Add/Delete/Modify bindings
5) Browse the networks host list (/etc/hosts)
6) Open a module interface section for editing (ControlCalc interfaces only)
7) Link/Compile/Write a MIL file on one or every host (if NFS is available)

3.1.4. Calling Interfaces from ControlCalc Applications

Up to this point, we have discussed only a mechanism for implementing message receipt and, optionally, reply to a message using the Polylith sink and function interfaces. It is equally important that ControlCalc programs be able to initiate message transfers, acting through Polylith source and client interfaces. Sink and function interfaces, as we saw, can be handled by treating the ControlCalc spreadsheet as a data base with computation on demand. Except for the computation itself, no programming beyond MIL-equivalent declarations are required. However, to implement source and client interfaces, interfaces must be invoked from within ControlCalc's functional programming language. To do this, one additional configuraton parameter is required: a declaration of which ControlCalc process implements the node.

3.1.4.1. Implementing Polylith Calls in ControlCalc

The basic approach we will take is to let the user access source and client interfaces as if they were functions in the spreadsheet, for the basic Polylith message passing functions mh_read and mh_write. The programmer would think of these interfaces as remote procedure calls, or as message passing calls. The function names would be the interface name. Because ControlCalc's language and compiler are, at this point, completely under the control of RTware, we can enhance the language to support dynamic function definitions such as this. Furthermore, ControlCalc's support for functional polymorphism relaxes the expected requirement of including the tool name as a parameter to the function. Without polymorphism this would always be necessary, since multiple instances of an interface name can exist, either because different modules use the same interface name or because a module is instantiated as a tool more than once. Access to the complete application and module MIL declarations allows the spreadsheet parser to see if the name is unique, and require an additional tool name parameter (which would have to be a constant string) only if the interface name is not unique within an application. Note that the compiler can determine from the MIL file how many parameters an interface declares. This means it is feasible, but perhaps not desirable for debugging, to distinguish between some common interface names by using differences in their parameter declarations and matching them to the function's parameter list. The only syntatic kludge required is that for client functions, output parameters would be required to be listed before input parameters (or vice-versa, by convention). Naturally, parameter ordering would have to match, since there is no parameter naming in the interface declarations.

Needless to say, it is also possible to provide direct implementation of Polylith's entire mh_ function set. Since ControlCalc supports text processing functions, it is reasonable to include all the select and query functions. ControlCalc's polymorphism inherently provides the varargs-style interface required for doing this.

3.1.4.1 Application Startup Method Conflicts Between Polylith and ControlCalc

As it currently stands, Polylith uses a static process model of programming which presents certain problems when being used with an interactive multi-processing system such as ControlCalc. The

primary restriction is that each module must be implemented as a process which can be executed by the Polylith bus program. This is essentially how the entire application is started. Each module process must be a uniquely named executable and cannot receive application level command line arguments. Polylith does provide an attribute declaration statement for the MIL module section, which allows an instantiated module program to read tagged attributes. This is essentially a way of passing arguments to a process (or else process arguments are essentially a way of defining process attributes, your choice).

ControlCalc, on the other hand, operates with a small set (just three at this time) of executable modules which are configured dynamically according to declarations and functional programs. An executable ControlCalc program is actually compiled by the spreadsheet process, with code placed in shared memory sections. Being a multi-tasking spreadsheet, multiple code segments can be created, each potentially operating on any part of the entire three-dimensional spreadsheet data space. When the application is run, an executive program is invoked for each code segment, and is passed the parameters it needs to link to shared memory, link to various resources used by the code, locate the code segment, and execute the code segment when triggered.

The source/client implementation described in the above section requires that each ControlCalc program segment process be the binary for a node. To do this requires changes to either Polylith or ControlCalc's process model.

Changing Polylith to accomodate ControlCalc processes would involve making the bus dynamically configurable. Rather than requiring that all nodes be executed by the bus, the bus could start independently and allow processes to attach later. In attaching, a process would pass its declarations to the bus, the bus would validate the declarations against unattached services in its MIL declarations, and if a valid match were found, the bindings would be instantiated. At that point, the process would be free to make calls on the node's interfaces. This approach would allow ControlCalc to set up the necessary resource declarations for a process which is declared as the implementation of a node. Those resources would be equivalent to the MIL file declarations, and would be passed in to the bus.

Changing ControlCalc to accomodate Polylith involves generating copies of the ControlCalc general-purpose code segment master process, with unique names incorporating both the template key and the code segment key. Parameters required by these process and the names of the binaries would have to be added to the module's MIL section. Since ControlCalc creates the code segment and establishes the process parameters when the application is started, the launch of an application would involve the following sequence:

1) Start all ControlCalc templates involved in the application.
2) Each ControlCalc template process generates the necessary binaries and modifies the relevent MIL sections.
3) Rendezvous on all ControlCalcs completing their generation phase.
4) Compile the master MIL file.
5) Start the bus.

While this process seems a bit involved, it can be made completely transparent to the user. We are considering that a second bus be used to simply handle the startup process. That bus could be torn down when the application is running.

There is a simpler approach possible, but it will result in a significant performance loss. The other approach is to use the single node process that handles input interfaces to handle output interfaces on behalf of the ControlCalc program. This would mean that each Polylith call from ControlCalc would involve an additional message passing transaction to the node process. We consider this to be unacceptable for real-time systems.

3.1.5. Virtual FIFO Connections

ControlCalc currently provides a TCP/IP socket I/O option that allows spreadsheets to exchange data between "history cells". ControlCalc history cells are actually complete first-in, first-out (fifo) queues, with a user configurable maximum size. They can be used internally as rotating ring buffers that remember the last X values (hence the name history cell), or for buffering communication with other ControlCalc processes of I/O agents, without requiring operating system intervention. The socket I/O option transfers data from the output side, dequeuing it from the output history cell, to the input side, enqueuing onto the input history cell. A block transfer size configuration parameter gives the user control over the effeciency of this process.

Implementing an equivalent option with Polylith is quite straightforward with the system we are describing. Note that this virtualization of the history cell across two spreadsheets is inherently declarative, requiring no procedural programming by the spreadsheet user. Therefore it could be made an option in the Polylith module definition utility. An exact equivalence to the current implementation should be possible by using a Polylith array data type and the direct connect option.

3.1.5. Desired Performance Enhancements

There is one feature we would like to add to Polylith to support the high performance networking usually required in distributed real-time applications. That is to make the "keep-alive" and "direct-connect" options configurable per interface. We think this could be done by establishing key-word object attributes which are used by the bus when the system is started. Object attribute statements like KEEP_ALIVE = interface_name or DIRECT_CONNECT = interface_name would instruct the bus to use these modes for bindings to the specified interfaces. They would also instruct ControlCalc to instantiate separate processes for interfaces using the direct-connect mode, or if more than the maximum number of interfaces allowed by the operating system are in the keep-alive mode. This approach would allow tuning the application to achieve a balance of performance and complexity.

3.2. Distributed Control System II: Distributed Code

The second phase of a ControlCalc-based DCS system will allow the code segment processes (known as scans) to operate on distributed and heterogeneous hosts. This is complicated by the fact that ControlCalc scans more closely resemble threads than processes. As such, they could be distributed across processors on a tightly coupled multi-processor platform running, for example, Mach. However, a DCS system is usually loosely coupled with a LAN network. For this reason, intelligent compiler technology is required to minimize the network traffic required to maintain coherence between distributed copies of the shared memory data space.

3.2.1. Advantages of DCS-II

The advantage of DCS-II is that no programming is required to explicitly pass messages between the different scans of the application. The ControlCalc spreadsheet programmer already interactively declares a scans operating mode, from a short list of choices and parameters. To run a scan remotely, only some additional parameters will be required in that declaration. These parameters would specify the host, the network port with any associated configuration parameters and possibly the network operating mode.

One of the key design goals is to preserve the fundamental advantages of the ControlCalc spreadsheet paradigm. In particular, the system must present on-line updates from all active nodes to the user-interface part of the spreadsheet. Right now, ControlCalc users can watch spreadsheet cell values changing in any scan in the system by simply looking at the cells which define the function

being executed. This is inherent in the shared memory design of the spreadsheet, since both the executing scans and the spreadsheet interface are using the same shared memory locations for the data. The user interface program is simply polling and displaying the most recent result at a speed (about 5 hz.) which is adequate for human reaction. This shared memory approach also allows the inclusion of custom graphical user interfaces by mapping the variables associated with graphics objects into the same shared memory areas. Manipulating graphical objects is thus done through a declarative interface which establishes bindings, and does not require any programmatic language support for the most common operations.

3.2.2. Implicit Message Passing

The most difficult part of the DCS-II design is the implicit message passing between the "threads" of a ControlCalc spreadsheet. ControlCalc's organization allows the user to declare any page of the three-dimensional spreadsheet to be a scan ("thread"). Each page can contain both function cells with both the expression and its result and data cells of various types. This organization lends itself well to the DCS system because the data on a page can be kept locally on a remote note along with the code for that page.

However, the spreadsheet model treats all data cells as globals in the sense that cells on one page can be read by expression cells on another. There are even cell write functions that do cell assignments by side effect. Note that except for the cell write functions, a spreadsheet expression writes its result to the data part of the expression cell itself, so by far the majority of writes are local.

The spreadsheet compiler will have to handle remote reads and writes, and should do so without requiring any explicit programming. The simplest method is to have the compiler convert all remote reads and writes to message calls, in-line at the point the executing function requires the access. This will maintain complete consistency when done with a bus such as Polylith that insures atomicity of these operations. The question is whether perform can enhanced using caching schemes.

A write-through cache approach would require each remote node to maintain a cache table of remote cells that are referenced by local code. This is easily done with the compiler, since scan location is declared before code is generated. There would be the overhead of locking the cache before reading the value, and writes would always involve message calls, but we anticipate a major reduction in message call traffic with this approach.

More sophisticated caching schemes may not be required for one simple reason: ControlCalc provides block copy functions which can operate between pages. This means that the programmer can optimize the application by understanding that it is much more efficient to explicitly move blocks of data between remote pages than to rely on the implicit message calling.

3.2.3. Cross Compilation

In order to run on heterogeneous targets, ControlCalc's internal compiler must be capable of generating code as a cross compiler. Currently there are three versions of the compiler, generating code for the 386/486, 68K and 88K processors. However, the compiler is not packaged separately, so it will have to be repackaged and supplied as a set of separate cross compilers. This work will be part of the SBIR Phase II proposal.

3.2.4. Target Platforms

A target platform for the DCS system impacts ControlCalc in 1) the cpu architecture, which requires a cross compiler, 2) the operating system and 3) the network interface. To this point, Polylith has

focussed on UNIX platform with TCP/IP networks. The C compilers are, of course, already available. In the real-time control area, however, platforms are much more heterogeneous.

DSP processor are greatly increasing in popularity for real-time applications, not just for classical signal processing but for their high-speed data acquisition and raw processing power. The NRL group is planning to use a DSP processor on a VME board. That board uses the ADSP-21020 processor from Analog Devices, which can achieve 100 peak and 66 sustained MFLOPS. This system also comes with a set of mezzanine I/O modules with, for example, a 12-bit, 10 Mhz. A/D convertor. The operating system with that board is SPOX, which is also widely available on commercail DSP processors such as those from National Instruments on PC-AT systems. SPOX is not UNIX, nor does it support TCP/IP for communication with the host computer. However, it does have a rich set of message passing functions, so it is probably feasible to run Polylith on SPOX.

In the general-purpose processor market, there are a number of popular real-time operating systems which have various levels of compatibility with UNIX and TCP/IP. ControlCalc currently uses the OS-9 system from Microware which supports the socket library. As part of our work with Polylith, we intend to validate that OS-9 can support Polylith. U.S. government agencies, including DoD and NASA have worked extensively with LynxOS and VxWorks. Of the two, LynxOS is closest to a realtime UNIX, while VxWorks is designed as a remote realtime kernel that uses UNIX for hosted development and supports the full TPC/IP protocol. The NRL project will probably specify LynxOS, due to its high POSIX conformance and self-hosted real-time capabilities.

Finally, a DCS application will include systems which are not real-time, particularly for functions like operator interface, configuration control and back-end processing. For these systems, the standard is UNIX running on RISC workstations. We are using a Sun Sparc system, since it is a de facto industry standard. We are also running an 88Open machine from Encore.

For Phase II, we will select a set of platforms which will demonstrate the range of capabilities and is very acceptable to the DoD community. Our decision at this point is driven by the NRL project toward Sparc, LynxOS, OS-9 and SPOX. A DCS system running across this set of platforms will be both an impressive demonstrative of the capabilities of Polylith/ControlCalc, and one that meets the specifications of many DoD projects.

4. Conclusion

At this point in Phase I of the project, we have a complete design for the first level Distributed Control System (DCS-I) based on the integration of Polylith and ControlCalc into a complete Module Interconnection Framework. The system lets multiple languages communicate between heteogeneous platforms. A major result will be integration of a functional language and application generator system represented by ControlCalc into a modular format. ControlCalc's unique capabilities as a control processor and data manipulation engine will therefore be able to interoperate smoothly with programs written in third generation languages such as 'C'. This will allow the transition between systems to be made only where it is appropriate, and in a modular, step-by-step manner that maximizes application reliability.

The second level, DCS-II system now has a fairly complete conceptual framework and a target environment that is attractive to many DoD projects, including the NRL. While requiring significantly more effort than DCS-I, DCS-II will be a much more transparent distributed environment for the more tightly integrated areas of a DCS application. The use of distributed threading within a common shared memory data set allows maximum performance while relieving the programmer of almost all the work involved in the distribution scheme.

The remainder of phase I will involve on-going test implementations of the concepts described in this report and the preparation of a detailed phase II proposal. Much of our work will focus on the NRL project as a good representative application area of the resulting system. We hope to have a demonstration system running between the Sun and an OS-9 system by the end of Phase I. This system will show a C program running on one platform working with a ControlCalc program on another, using the Polylith software bus in a Module Interconnection Framework.